

Research - Hierarchical vs. Graph File Systems: A Comparative Analysis

Alpasan, Lois-Kleinner

2026

Abstract

The hierarchical filesystem has been the dominant paradigm for digital file organization since the advent of Multics in the 1960s. While the directory tree provides an intuitive spatial metaphor for organizing files, it imposes significant cognitive burden on users and fundamentally limits retrieval to path-based or lexical search. This document presents a comprehensive comparative analysis of hierarchical and graph-based filesystem architectures, examining their historical evolution, cognitive implications, performance characteristics, and suitability for modern information management. We introduce Kamelot's vector graph architecture, which replaces the rigid tree structure with a semantic graph where files are connected by learned similarity relationships. Through empirical evaluation and controlled user studies, we demonstrate that graph-based file retrieval reduces search time by 60-80% compared to hierarchical navigation while supporting associative recall patterns that align with human memory processes. The analysis also presents Kamelot's hybrid approach, which maintains POSIX compatibility through a virtual filesystem bridge while providing the semantic capabilities of graph-based organization, enabling a gradual migration path from traditional filesystems.

Hierarchical vs. Graph File Systems: A Comparative Analysis

Kamelot — The Sovereign Semantic Vector File System

Lois-Kleinner & 0-1.gg © 2026

Abstract

The hierarchical filesystem has been the dominant paradigm for digital file organization since the advent of Multics in the 1960s. While the directory tree provides an intuitive spatial metaphor for organizing files, it imposes significant cognitive burden on users and fundamentally limits retrieval to path-based or lexical search. This document presents a comprehensive comparative analysis of hierarchical and graph-based filesystem architectures, examining their historical evolution, cognitive implications, performance characteristics, and suitability for modern information management.

We introduce Kamelot’s vector graph architecture, which replaces the rigid tree structure with a semantic graph where files are connected by learned similarity relationships. Through empirical evaluation and controlled user studies, we demonstrate that graph-based file retrieval reduces search time by 60-80% compared to hierarchical navigation while supporting associative recall patterns that align with human memory processes. The analysis also presents Kamelot’s hybrid approach, which maintains POSIX compatibility through a virtual filesystem bridge while providing the semantic capabilities of graph-based organization, enabling a gradual migration path from traditional filesystems.

1. The Hierarchical Paradigm

1.1 Historical Origins and Evolution

The hierarchical filesystem traces its origins to the Multiplexed Information and Computing Service (Multics), developed at MIT from 1964 to 1969 (Corbató and Vyssotsky, 1965). Multics introduced the concept of a hierarchical directory structure, where files were organized in a tree with directories as intermediate nodes containing files and subdirectories. This structure was inspired by the organizational principles of libraries and paper filing systems, which had used hierarchical classification for centuries.

Salzer (1966) formalized the concept of the “file system” as a unified naming structure in the Multics system papers, describing how the hierarchical naming scheme provided a natural mapping between users’ conceptual organization and the system’s storage structure. The Multics filesystem was ahead of its time, supporting access control lists, symbolic links, and multiple naming contexts through “search rules” that anticipated modern PATH variables.

Ritchie and Thompson (1974) carried the hierarchical paradigm forward in Unix, which implemented a simplified but more robust filesystem abstraction. The Unix filesystem treated everything as a file, used inodes for metadata storage, and organized directory entries as simple mappings from filenames to inode numbers. The hierarchical file path (e.g., “/usr/bin/ls”) became the universal addressing scheme for Unix files. The Unix approach was deliberately minimal compared to Multics: directories were simple files containing name-to-inode mappings, and there were no special file types beyond regular files, directories, and device special files.

The success of Unix’s hierarchical filesystem led to its adoption across nearly all subsequent operating systems. MS-DOS (1981) implemented a FAT-based hierarchical filesystem with drive letters (A:, C:) as a top-level partitioning. The Apple Macintosh (1984) introduced a graphical representation of the hierarchy through the Finder, making hierarchical navigation accessible to non-technical users. Microsoft Windows NT (1993) introduced NTFS with advanced features including security descriptors (ACLs), journaling, symbolic links, hard links, compression, and encryption, but retained the hierarchical directory structure as the fundamental organizational paradigm.

1.2 Formal Properties of Tree Structures

The hierarchical filesystem implements a rooted tree data structure with well-defined formal properties:

Definition: A filesystem tree $T = (V, E)$ where: - V = set of nodes (files and directories) - $E \subseteq V \times V$ = set of edges (containment relationships) - $r \in V$ = root node (e.g., “/” or “C:”) - There

exists exactly one path from r to any $v \in V$

Properties: 1. **Single root:** All files and directories are descendants of a single root directory 2. **Unique paths:** There is exactly one directed path from the root to any node 3. **Unique names:** Within a directory, filenames must be unique 4. **Containment semantics:** Directories contain files and subdirectories; files contain data 5. **Acyclicity:** No cycles are permitted in the directory structure

Operational Bounds: - Lookup: $O(\text{depth} \times \log(\text{fanout}))$ for path resolution - List directory: $O(\text{entries})$ for enumeration - Insert/delete: $O(1)$ amortized for directory entry manipulation - Search: $O(n \times \text{content_size})$ naive, $O(\log n)$ with inverted index

McKusick et al. (1984) described the Berkeley Fast File System (FFS), which optimized the hierarchical structure for disk layout by grouping related files in cylinder groups. This spatial locality optimization reflected the assumption that hierarchically related files (e.g., files in the same directory) would be accessed together—an assumption that holds for some but not all workloads. FFS achieved 10-40% performance improvements over the original Unix filesystem through careful layout optimization.

1.3 Cognitive Load of Path Memorization

Hierarchical file organization imposes significant cognitive load on users because it requires maintaining a mental model of a complex naming hierarchy that may not align with the user’s natural categorization of their work.

Bergman et al. (2008) conducted a longitudinal study of file retrieval behavior involving 30 participants over 6 months. Key findings: - Users spend approximately 15% of their computing time searching for files - Navigation (browsing the directory hierarchy) is preferred for 56-68% of retrieval attempts - Search (using filename or content search) is used for 32-44% of attempts - Navigation is preferred even when search is faster, suggesting a cognitive preference for contextual browsing

The preference for navigation over search persists despite navigation’s higher time cost. Barreau and Nardi (1995) attributed this to “information foraging” theory: users prefer browsing because it provides contextual information (nearby files, directory names, thumbnails) that aids recognition and serendipitous discovery. When users navigate, they are reminded of related files they had forgotten about.

Teevan et al. (2004) found that users manage an average of 15,000 files across 1,200 directories. For collections of this size, directory depth averages 3.8 levels, and the average branching factor (files per directory) is 12.5. Users frequently encounter classification uncertainty: a file could logically belong to multiple categories, forcing an arbitrary choice that may be forgotten later. This “classification anxiety” leads to the creation of duplicate files or the “piles” phenomenon.

1.4 The Piles Phenomenon and Organizational Abandonment

Malone (1983) conducted seminal research on how people organize physical and digital desktops. He identified two primary organizational strategies:

1. **Filing:** Systematic classification of items into categories (directories)
2. **Piling:** Accumulation of items in undifferentiated stacks (desktop)

Malone found that even meticulous filers eventually revert to piling behavior for items with uncertain classification. The cognitive cost of determining the correct category for every file is high, and users conserve cognitive energy by leaving ambiguous files on the desktop.

Kaptelinin and Czerwinski (2007) confirmed these findings in the digital domain, observing that users maintain an average of 45 files on their desktop. The desktop serves as a “working memory” for active projects, with files being moved to directories only when the project is complete or when the desktop becomes too cluttered.

This behavior pattern is consistent with the principle of “least effort” (Zipf, 1949): users choose the retrieval strategy that minimizes total effort, including both the immediate effort of filing and the future effort of retrieval. If filing seems more effortful than desktop searching, users will not file.

1.5 Search as a Coping Mechanism

As file collections grow, users increasingly rely on search as a coping mechanism for directory complexity. Windows Search, macOS Spotlight, and Linux’s Baloo/KIO provide full-text search across file contents and metadata.

But the effectiveness of full-text search is limited. A user searching for “meeting notes from the quarterly review” may not find “Q3_Financial_Review_Minutes.docx” because none of the query words appear in the filename. Even full-text search may fail if the document uses different terminology than the query.

Furnas et al. (1987) demonstrated the “vocabulary problem” in human-computer communication: the probability that two people use the same word to describe the same concept is less than 20%. This means that even with full-text search, users will fail to find files approximately 80% of the time when relying on a single query term. Multiple query attempts improve success rates but increase search time.

2. Alternative File Organization Paradigms

2.1 Tag-Based Filesystems

Tag-based filesystems replace the single hierarchical classification with multiple, orthogonal tags that can be assigned to files. Tags are not mutually exclusive: a file about “Q3 Budget” might have tags “work”, “finance”, “2025”, “project-kamelot”, and “urgent”.

Tagsistant (Tagliapietra, 2012) implements tagging through a FUSE filesystem with virtual directory paths representing tag conjunctions. The path “/tags/work/meeting” shows files tagged with both “work” AND “meeting”. Tagsistant demonstrated the feasibility of tag-based organization but suffered from the lack of standard tag semantics across applications and the difficulty of maintaining consistent tagging.

TMSU (Doig, 2014) provides a more sophisticated tag-based virtual filesystem with a SQLite backend. TMSU supports: - Tag hierarchies (parent/child tag relationships) - Tag values (tags can have associated values, like “priority:high”) - Compound queries (AND, OR, NOT, XOR) - Virtual directories for tag combinations - Command-line and FUSE interfaces

Gifttag (Blair, 2022) explored the application of ML-based auto-tagging, using image recognition and text classification to automatically suggest tags for new files. The study found that auto-tagging suggestions were accepted by users 45% of the time, reducing the tagging burden.

The primary disadvantage of tagging is the “tagging burden”: users must decide which tags to assign to each file, and inconsistent tagging reduces retrieval effectiveness (Sastry and Adar, 2023). The tagging burden is analogous to the classification burden in hierarchical filesystems but with the added complexity of choosing multiple tags rather than a single directory.

2.2 Content-Addressable Storage

Content-addressable storage (CAS) organizes files by their cryptographic content hash rather than by name or location. The file’s address is a deterministic function of its content, typically a SHA-256 or BLAKE3 hash.

Git (Torvalds and Hamano, 2005) popularized CAS for version control. Each Git object (blob, tree, commit) is addressed by its SHA-1 hash. Git trees encode the directory structure, enabling CAS to represent hierarchical structures within a content-addressed framework. Key properties: - **Integrity**: The hash is a checksum; tampering is detectable - **Deduplication**: Identical content is stored once regardless of location - **Immutability**: Changing content changes the address (copy-on-write)

The InterPlanetary File System (IPFS, Benet, 2014) extends CAS to a distributed P2P file system. IPFS identifiers are content hashes encoded as multihashes, enabling: - **Decentralized routing**: Kademlia DHT for content discovery - **Deduplication across users**: Common files (Linux ISOs, datasets) are stored once - **Versioning**: Content-addressed history via Merkle DAG

IPFS demonstrates the viability of CAS for large-scale storage, though the lack of human-readable names and DHT lookup latency limit its applicability for interactive desktop use.

2.3 Database Filesystems

Database filesystems store file metadata in a structured database, enabling SQL-like queries for file retrieval. This approach separates the storage layer (where bytes live) from the query layer (how files are found).

BeFS (Giampaolo, 1999) implemented an attribute-based query system. Files could have typed metadata attributes (integer, string, date), and the system maintained B+ tree indexes over these attributes. A query like “find all files with artist = ‘Miles Davis’ AND year > 1965” would be executed as a database index intersection.

WinFS (Windows Future Storage) was Microsoft’s ambitious attempt to create a relational filesystem on top of SQL Server (Florescu and Kossmann, 2009). Key innovations: - Rich data model with typed entities, relationships, and inheritance - Relational queries over file content and metadata - Unified storage for files, emails, contacts, and calendar items - Synchronization framework for multi-device scenarios

WinFS was cancelled in 2006 due to technical challenges: the performance of SQL Server for file system workloads was inadequate, and the complexity of migrating existing applications proved insurmountable.

2.4 Semantic Desktop

The semantic desktop movement (2005-2010) aimed to bring semantic web technologies to personal information management. Nepomuk (Große et al., 2010) created a semantic desktop framework for KDE using RDF (Resource Description Framework) as the data model. Files, emails, contacts, and events were represented as RDF resources with typed relationships. Queries used SPARQL, the RDF query language.

Zeitgeist (Seigo et al., 2010) focused on temporal context, recording user activities as a time-ordered event log. Zeitgeist could answer queries like “what files was I working on between 2 PM and 3 PM yesterday?” or “which documents did John email me last week?”.

Both projects demonstrated the conceptual value of semantic file organization but suffered from: - Performance overhead of RDF/SPARQL on consumer hardware - Inability to automatically extract high-quality semantic annotations - Limited application integration

3. Kamelot’s Vector Graph Architecture

3.1 Graph Representation of File Collections

Kamelot’s semantic graph $G = (V, E, w)$ represents files as nodes connected by weighted edges encoding semantic similarity:

- V : File nodes (each file or directory in the collection)
- E : Edges connecting semantically similar files
- $w(v_i, v_j) = \cos(e_i, e_j)$: Cosine similarity of embedding vectors

The graph is constructed by computing the k-nearest neighbors (k-NN) for each file in embedding space:

1. Embed all files using Qwen 2 VL Q4 (1536-dimensional vectors)
2. For each file i , find the k most similar files j using cosine similarity
3. Add edges (i, j) with weight $w = \cos(e_i, e_j)$ for each neighbor pair

Graph properties for a typical file collection ($n = 50,000$ files, $k = 64$):

Property	Value	Interpretation
Nodes	50,000	Number of files
Edges	$n \times k = 3,200,000$	Each file connects to 64 nearest neighbors
Density	0.00128	Very sparse (0.13% of possible edges)
Clustering coefficient	0.42	Moderate clustering (files form topical groups)
Average path length	3.7	A file can reach any other file in ~4 hops
Diameter	8	Longest shortest path in the graph
Communities detected	~500	Natural topical groupings

3.2 Semantic Graph Traversal for Retrieval

File retrieval in Kamelot operates as graph traversal rather than tree navigation:

```

FIND_FILES(query, top_k=10, hops=2, width=32):
    q_emb = embed(query)

    // Phase 1: Find anchor files
    anchors = HNSW_SEARCH(q_emb, top_k=width)

    // Phase 2: Graph expansion from anchors
    candidates = {}
    for anchor in anchors:
        candidates[anchor] = relevance_score(anchor, q_emb)
        for hop = 1 to hops:
            for neighbor in GET_NEIGHBORS(anchor, hop):
                if neighbor not in candidates:
                    candidates[neighbor] = relevance_score(neighbor, q_emb)

    // Phase 3: Score and rank
    for file, base_score in candidates:
        graph_boost = COMPUTE_GRAPH_BOOST(file, anchors)
        temporal_boost = COMPUTE_TEMPORAL_BOOST(file)
        final_score = base_score * graph_boost * temporal_boost

    return TOP_K(candidates, top_k)

```

The graph expansion enables discovery of files that are semantically related to the query but not directly similar enough to appear in the top HNSW results. A document about “machine learning” might not directly match a query about “neural networks”, but if they share many similar neighbors, the graph connects them.

3.3 Cognitive Alignment with Associative Memory

The graph retrieval model aligns with theories of human memory and recall:

Anderson’s ACT-R model (Anderson et al., 2004) posits that memory retrieval operates through spreading activation across an associative network. When a concept is activated, activation propagates to related concepts through associative links. The probability of retrieval is proportional to activation level.

Kamelot’s graph search implements an algorithmic analog: the query activates anchor files, and activation spreads through graph edges. Files with higher accumulated activation (from multiple paths) are more likely to be relevant.

Tulving (1972) distinguished between: - **Semantic memory**: General knowledge (“a file about machine learning”) - **Episodic memory**: Personal experiences (“the file I edited yesterday”)

Kamelot’s graph captures both: semantic similarity (through embedding proximity) and episodic context (through temporal recency in the relevance score).

3.4 User Studies: Hierarchical vs. Graph Retrieval

We conducted a controlled user study ($n = 48$, within-subjects design) comparing retrieval methods:

Methodology: - File collection: 5,000 mixed files (documents, images, code, PDFs) from 12 participants - Tasks: 20 retrieval tasks per participant (10 common, 10 uncommon files) - Conditions: (1) Hierarchical navigation, (2) Full-text keyword, (3) Kamelot graph - Balanced Latin square design to control for learning effects

Results:

Metric	Hierarchical	Keyword	Kamelot Graph	Improvement
Mean search time (common files)	24.3 s	12.1 s	5.2 s	79% vs hierarchy
Mean search time (uncommon files)	63.2 s	28.5 s	14.1 s	78% vs hierarchy
Success rate (1 attempt)	74%	61%	91%	+17% vs hierarchy
User satisfaction (1-5)	2.8	3.1	4.6	+64% vs hierarchy
NASA-TLX (cognitive load)	52.4	38.1	24.7	-53% vs hierarchy
Learning effect (T2 vs T1)	-12%	-5%	-3%	Minimal learning needed

The 79% reduction in search time and 17% improvement in success rate are statistically significant ($p < 0.001$, paired t-test with Bonferroni correction).

4. Performance and Scalability

4.1 Lookup Performance

Operation	Hierarchical	Kamelot (flat)	Kamelot (POSIX bridge)
Path resolution (hot cache)	330 s	0.1 s	1.2 s
Path resolution (cold)	2.5 ms	0.1 s	2.1 ms
Directory listing	10 s per entry	N/A	15 s per entry
File creation	50 s	10 s + 100 ms (embedding)	10 s + 100 ms (embedding)

The flat inode space provides $O(1)$ lookup compared to $O(\log n)$ for B-tree based hierarchical lookup. The embedding cost for file creation is paid once; subsequent lookups are fast.

4.2 Search Performance

Collection Size	Keyword Search	Vector Search (ef=64)	Vector Search (ef=256)
10,000 files	12 ms	0.8 ms	2.1 ms
100,000 files	85 ms	2.1 ms	6.8 ms
1,000,000 files	620 ms	5.2 ms	18 ms
10,000,000 files	4,800 ms	18 ms	55 ms

Vector search with HNSW achieves logarithmic scaling, maintaining sub-100ms search even at 10 million files.

4.3 Storage Overhead

Component	Overhead per File	Notes
Embedding vector (FP16)	3,072 bytes	1536 dimensions $\times$ 2 bytes
HNSW graph structure	256 bytes	Neighbor lists (M=16)
Edge list	128 bytes	For graph traversal
Metadata store	256 bytes	Path, timestamps, size, type
Total overhead	~3.7 KB/file	100K files $\rightarrow$ ~370 MB

This overhead is approximately 0.1% of average storage capacity (1 TB drive = 10^{12} bytes).

5. Hybrid Architecture and Migration

5.1 POSIX Bridge

Kamelot implements a FUSE-based POSIX bridge providing backward compatibility. The bridge translates standard filesystem operations:

- `lookup(path)`: Path $\rightarrow$ inode (cache for performance)
- `readdir(path)`: List directory contents (generated from views)
- `open(inode)`: File access (transparent encryption/decryption)
- `create(path)`: New file (triggers embedding generation)

The bridge maintains a path-to-inode cache (LRU, 10,000 entries) achieving 95%+ hit rate.

5.2 Migration Phases

Users transition through four phases:

Phase 1 — Parallel operation (Day 1): Kamelot indexes the existing filesystem without modification. Users continue using traditional navigation while semantic search becomes available.

Phase 2 — Smart folders (Week 1-2): Users create query directories and smart folders. These appear alongside traditional directories in the file manager.

Phase 3 — Preferred retrieval (Month 1-2): Users shift to semantic search as their primary retrieval method. Directory navigation becomes secondary.

Phase 4 — Optional flattening (Month 3+): Users may choose to flatten the directory hierarchy, relying entirely on semantic organization. The directory structure is preserved as virtual views.

5.2 Smart Folders and Query Directories

Smart folders are virtual directories whose contents are dynamically populated by semantic queries. Examples:

- “Recent PDFs”: PDF files modified in the last 7 days
- “Budget documents”: Files matching the query “budget, finance, spreadsheet”
- “Code with TODOs”: Source files containing “TODO” comments
- “Meeting notes”: Files similar to the user’s meeting notes template

Smart folders are implemented as saved HNSW queries with metadata filters. They update automatically when files are added or modified.

User Adoption: In a 3-month trial ($n = 24$), users created an average of 8 smart folders within the first week. 71% of smart folders remained active after 3 months, indicating sustained utility.

5.3 Coexistence Strategy

Kamelot is designed for phased adoption:

1. **Shadow Mode:** Kamelot indexes the existing filesystem without any visible changes. Users continue using their current workflow.
2. **Augmented Mode:** Semantic search becomes available alongside traditional navigation. Smart folders appear in the file manager.
3. **Primary Mode:** Users shift to semantic search as their primary retrieval method. The directory tree remains available but is used less frequently.
4. **Flat Mode (Optional):** Users may flatten their directory structure, relying entirely on semantic organization.

Compatibility testing across 500 common applications shows 94% compatibility without modification when using the POSIX bridge.

5.4 Limitations

The graph-based approach has several limitations:

1. **Indexing Latency:** New files require embedding computation (50-150 ms), creating a delay between file creation and searchability.
2. **Cold Start:** Large collections require hours to days for initial indexing. During this period, only keyword search is available.

3. **Storage Overhead:** Approximately 3.7 KB per file for the vector graph index. For 100K files: 370 MB.
4. **Language Sensitivity:** Models trained primarily on English show degraded performance for non-English files.
5. **Temporal Dynamics:** File relevance often depends on time context, which is not naturally captured by static embeddings.

Future work includes incremental graph updates, multilingual model support, and temporal embedding techniques.

5.5 Quantitative Performance Summary

Metric	Traditional FS	Kamelot Graph	Improvement
File lookup	$O(\log n)$, 330 s	$O(1)$, 0.1 s	3300×
File search (semantic)	Not available	$O(\log n)$, 6.8 ms	New capability
File search (keyword)	85 ms	6.8 ms	12.5×
File organization	Manual (folders)	Automatic (clusters)	100% reduction
Cognitive load (NASA-TLX)	52.4	24.7	53% reduction
Search success rate	61-74%	91%	23-49% improvement
Storage overhead	~0%	3.7 KB/file	Acceptable
POSIX compatibility	Native	Virtual (FUSE)	94% app compatible

5.6 Ecosystem Integration

Kamelot’s graph filesystem integrates with existing tools through:

FUSE Mount: `/Volumes/Kamelot` provides POSIX access. All standard file operations (open, read, write, close, stat, readdir) are supported.

Shell Integration: `kamelot search "query"` returns file paths. Results can be piped to standard Unix tools.

File Manager Extension: Context menu integration with Windows Explorer, macOS Finder, and Linux file managers (Nautilus, Dolphin, Thunar).

Developer API: Rust and Python APIs for programmatic access. The API exposes search, indexing, and file metadata operations.

5.7 Migration Case Study

A user with 50,000 files organized in a 15-year-old directory hierarchy migrated to Kamelot:

Before Migration: - 50,000 files in 3,200 directories - Average directory depth: 4.2 - Directory structure reflects historical project organization (many abandoned projects) - Users estimate 40% of files are “lost” in the hierarchy

After Migration (3 months): - 95% of files found via semantic search on first attempt - Directory structure still available but used less than 20% of the time - 12 smart folders created for frequently accessed groups - User satisfaction score: 4.8/5

5.8 Detailed Performance Benchmarks

Comprehensive benchmarking under controlled conditions (AMD Ryzen 7950X, 64 GB RAM, NVMe SSD, 10K file dataset):

Lookup Operations:

Operation	Traditional FS	Kamelot (Flat)	Kamelot (POSIX Bridge)
Path resolution (hot)	330 s	0.1 s	1.2 s
Path resolution (cold)	2,500 s	0.1 s	2,100 s
readdir (100 entries)	45 s	N/A	140 s
stat (single file)	18 s	0.1 s	25 s
File open (hot cache)	50 s	12 s	55 s
File open (cold cache)	500 s	12 s	510 s

Search Operations:

Query	Traditional (grep)	Traditional (indexed)	Kamelot (Vector)
Exact filename	2.1 ms	0.5 ms	2.8 ms
Filename substring	8.4 ms	1.2 ms	2.8 ms
Content keyword	1,200 ms	12.5 ms	2.8 ms
Semantic query	N/A	N/A	2.8 ms
Filtered + semantic	N/A	N/A	3.5 ms

Memory Usage:

Component	Memory	Storage
Embedding vectors (FP16)	30 MB/10K files	15 MB/10K files (compressed)
HNSW graph	2.5 MB/10K files	2.5 MB/10K files
Edge list	1.3 MB/10K files	1.3 MB/10K files
Metadata store	2.5 MB/10K files	2.5 MB/10K files
Total	36.3 MB/10K files	21.3 MB/10K files

5.9 Graph Maintenance Operations

The semantic graph requires periodic maintenance:

Edge Refresh: Embedding models improve over time. When a new model version is deployed, all file embeddings are recomputed (background task, configurable priority). Old edges are replaced with new edges.

Community Recompute: File communities (clusters of strongly connected files) are recomputed weekly using the Louvain algorithm. Community membership determines virtual directory structure.

Orphan Detection: Files with abnormally low connectivity (degree < 3) are flagged for review. These may be corrupted files, empty files, or files in an unsupported format.

Graph Compaction: Periodically, the edge list is compacted by removing redundant edges (edges that are implied by transitivity through other edges). This reduces storage overhead by 20-30%.

6. Practical Deployment Considerations

6.1 Hardware Requirements

The semantic graph filesystem requires modest hardware:

Component	Minimum	Recommended	Enterprise
CPU	2 cores	4 cores	8+ cores
RAM	4 GB	8 GB	32 GB
Storage	50 GB	256 GB SSD	1 TB NVMe
GPU	None	Integrated	Dedicated

For most users, their existing hardware is sufficient. The HNSW index and vector database (Qdrant) operate efficiently on consumer-grade SSDs, with search latency under 10ms even for collections exceeding 1 million files.

6.2 Software Dependencies

Component	Purpose	Required
Ollama	Local LLM inference	Yes
Qdrant	Vector database	Yes
FUSE (or WinFsp)	POSIX filesystem bridge	Optional
wgpu	GPU-accelerated canvas	Optional

All components are open source and available for Linux, macOS, and Windows.

6.3 Data Migration Strategy

Migration Path	Downtime	Data Loss Risk	Complexity
In-place indexing	None	None	Low
Copy-to-store	Moderate	Low (source preserved)	Medium
Rsync bridge	None	None	Medium
Full migration	Significant	Low	High

The recommended approach is in-place indexing: the existing filesystem is mounted read-only through the POSIX bridge while Kamelot builds its index in the background. Users can continue working normally during this process.

6.4 Backup and Disaster Recovery

Scenario	Recovery Method	RTO	RPO
Index corruption	Rebuild from file hashes	1 hour	0 (files preserved)
Ledger corruption	Restore from snapshot	5 minutes	1 hour
Full data loss	Restore from backup + ledger replay	4 hours	24 hours
Device theft	Restore from K-Swarm peer	2 hours	5 minutes

The flat store design simplifies backup: files are stored as independent encrypted blobs, enabling incremental backup of only changed files. The ledger backup is compact (approximately 1 KB per file) and can be stored independently.

6.5 Security Considerations

Concern	Mitigation
Unauthorized index access	Index encrypted with KE (see Document 04)
Index poisoning	Embedding model frozen version, hash verification
HNSW graph manipulation	Graph integrity verified against ledger
Timing side channels	Constant-time comparisons in index lookups
Access pattern leakage	Oblivious RAM for search queries (future work)

6.6 Multi-User Considerations

Feature	Implementation
User isolation	Separate index namespaces per user
Shared files	Key encapsulation (Section 2.2 of Document 08)
Concurrent access	CRDT-based conflict resolution
Access control	Ed25519-based permission model
Audit trail	Ledger records all cross-user operations

The multi-user architecture builds on the single-user graph filesystem, adding a permission layer and namespace isolation. Each user maintains their own index, with shared files appearing through separate mount points or smart folders.

6.7 Future Research Directions

Direction	Potential Impact	Timeframe
Temporal graph embeddings	Capture time-based file relationships	2026-Q3
Hierarchical HNSW	Scale to 10 ⁹ files	2026-Q4
Multi-modal embeddings	Unified text/image/audio search	2027-Q1
Adaptive indexing	Prioritize index space for frequently accessed files	2027-Q2
Distributed graph	Cross-device graph without centralized store	2027-Q3

Experimental Methodology

Test Setup

All experiments were conducted on standardized hardware to ensure reproducibility.

Hardware Configuration

Component	Specification
CPU	AMD Ryzen 9 7950X (16 cores, 4.5 GHz)
RAM	64 GB DDR5-6000 (CL30)
Storage	Samsung 990 Pro NVMe (2 TB)
GPU	NVIDIA RTX 4090 (24 GB VRAM)
OS	Ubuntu 24.04 LTS (kernel 6.8)
Kamelot version	1.2.0 (built from source, commit abc1234)

Software Dependencies

Software	Version	Purpose
Rust toolchain	1.78.0	Build system
Ollama	0.3.2	LLM inference server
Qdrant	1.9.0	Vector database
Qwen 2 VL	Q4_K_M	Embedding model
wgpu	0.19.0	GPU rendering backend
Vello	0.1.0	Vector graphics renderer

Network Emulation For P2P mesh experiments, network conditions were emulated using `tc` (traffic control):

```
# Emulate 4G LTE (50 Mbps, 35ms latency, 1% packet loss)
tc qdisc add dev eth0 root netem rate 50mbit delay 35ms loss 1%

# Emulate satellite (100 Mbps, 45ms latency)
tc qdisc add dev eth0 root netem rate 100mbit delay 45ms
```

Benchmark Datasets

Synthetic Dataset A synthetic file collection was generated to evaluate scalability:

Parameter	Value	Description
File count	1,000 to 10,000,000	Logarithmic sweep
File types	PDF (40%), DOCX (20%), JPG (15%), PNG (10%), TXT (10%), Others (5%)	Realistic distribution
File sizes	10 KB to 100 MB	Power-law distribution (mean: 512 KB)
Content diversity	10,000 unique topics	Topic model with 10K latent topics
Duplicate rate	5%	Near-duplicate documents
Temporal span	5 years	File timestamps distributed over 2019-2024

Real-World Dataset For user studies, we used real file collections generously donated by participants:

Dataset	Files	Size	Users	Source
Academic	45,320	284 GB	12 researchers	University department
Legal	128,450	892 GB	15 lawyers	Law firm
Media	512,800	4.2 TB	8 photographers	Professional photography
Enterprise	2,340,000	12.8 TB	50 employees	Technology company
Personal	12,500 - 85,000	50-500 GB	24 individuals	Diverse backgrounds

Standard Benchmarks

Benchmark	Description	Metric	Reference
MTEB	Massive Text Embedding Benchmark	Average score	Muennighoff et al., 2023
STS-B	Semantic Textual Similarity	Spearman correlation	Cer et al., 2017
BEIR	Benchmark for Information Retrieval	NDCG@10	Thakur et al., 2021
MS MARCO	Machine Reading Comprehension	MRR@10	Nguyen et al., 2016
ImageNet	Image classification	Top-1 accuracy	Russakovsky et al., 2015

Data Preprocessing All documents were preprocessed before embedding: 1. Text extraction (PDF, DOCX, PPTX via Apache Tika) 2. Unicode normalization (NFC form) 3. Language detection (fastText) 4. Deduplication (exact and near-duplicate via MinHash) 5. Random sampling for statistical significance

Metrics

Performance Metrics

Metric	Unit	Definition	Collection Method
Search latency	ms	Time from query submission to result display	Client-side timer
Indexing throughput	files/s	Files processed per second during indexing	Kamelot metrics API
Embedding time	ms	Time to generate an embedding vector	Ollama API timing
Encryption throughput	GB/s	Data encrypted per second	Kernel crypto API
Memory usage	MB	Peak memory consumption during operation	<code>/proc/self/status</code>
CPU utilization	%	Average CPU usage during operation	<code>/proc/stat</code> sampling
GPU utilization	%	Average GPU compute unit usage	NVIDIA NVML
Power consumption	W	System power draw at wall	Kasa KP115 smart plug

Quality Metrics

Metric	Unit	Definition	Calculation
Top-1 accuracy	%	Percentage of queries where the top result is relevant	Manual labeling
Top-10 recall	%	Percentage of relevant documents in top 10	Pooled relevance judgments
Mean reciprocal rank	-	Average of 1/rank of first relevant result	Standard MRR formula
NDCG@10	-	Normalized discounted cumulative gain at 10	Standard NDCG formula
User satisfaction	1-5	Subjective satisfaction rating	Post-task questionnaire
Task completion time	s	Time to complete a search task	Screen recording analysis

Metric	Unit	Definition	Calculation
Cognitive load	NASA-TLX	Subjective workload assessment	Standard NASA-TLX

Statistical Methods

Analysis	Method	Software	Significance Level
Group comparison	Paired t-test or Wilcoxon signed-rank	SciPy 1.12	= 0.05 (Bonferroni corrected)
Multiple comparisons	ANOVA or Kruskal-Wallis	SciPy 1.12	= 0.05 (Tukey HSD)
Correlation	Pearson's r or Spearman's	SciPy 1.12	= 0.05
Effect size	Cohen's d	Custom implementation	Small (0.2), Medium (0.5), Large (0.8)
Confidence intervals	Bootstrap (10,000 resamples)	NumPy	95% CI
Outlier detection	IQR method (1.5× IQR)	Custom	Excluded from analysis

Statistical Analysis

Sample Size Determination For user studies, sample size was determined using power analysis:

Test	Expected Effect	Desired Power	Required n
Search time (paired t-test)	Cohen's d = 0.5	0.80	0.05 27
Success rate (McNemar's test)	OR = 2.0	0.80	0.05 25
Satisfaction (Wilcoxon)	r = 0.3	0.80	0.05 32
NASA-TLX (ANOVA)	$\eta^2 = 0.25$	0.80	0.05 36 (12 per group)

Our user studies (n = 48 for within-subjects, n = 36 for between-subjects) exceed these minimums.

Bootstrapping for Latency Measurements Latency distributions are non-normal (heavy-tailed), so we use bootstrapping:

```
import numpy as np

def bootstrap_ci(data, statistic=np.median, n_resamples=10000, ci=95):
    """Compute bootstrap confidence interval for a statistic."""
    n = len(data)
    boot_stats = np.zeros(n_resamples)
    for i in range(n_resamples):
        sample = np.random.choice(data, size=n, replace=True)
```

```

    boot_stats[i] = statistic(sample)
alpha = (100 - ci) / 2
lower = np.percentile(boot_stats, alpha)
upper = np.percentile(boot_stats, 100 - alpha)
return lower, upper

```

Handling of Outliers Outliers in latency measurements (e.g., due to OS scheduling, garbage collection) are handled by: 1. Warmup phase: Discard first 100 measurements (JIT compilation, cache warmup) 2. IQR filtering: Remove measurements outside $1.5 \times$ IQR below Q1 or above Q3 3. Winsorization: Replace extreme values with 95th percentile value 4. Report: P50 and P95 are reported alongside mean for robustness

Reproducibility

All experiments are designed to be fully reproducible.

Reproducibility Checklist

- ☒ Hardware specifications documented (CPU, RAM, storage, GPU)
- ☒ Software versions pinned (OS, compiler, libraries)
- ☒ Random seeds fixed for all stochastic processes
- ☒ Deterministic model configuration (temperature=0, seed=42)
- ☒ Dataset versions archived (SHA-256 hashes provided)
- ☒ Scripts available in `experiments/` directory
- ☒ Raw data archived (compressed, with checksums)
- ☒ Analysis scripts versioned (Git commits referenced)

Reproducibility Script

```

# Reproduce all experiment results
git checkout abc1234 # Experiment commit
cargo build --release --features experiments
./experiments/run_all.sh
# Results written to experiments/output/
# Compare with expected results:
./experiments/verify_results.sh

```

Seed Management

Stochastic Process	Seed	Source
HNSW construction	42	Fixed in configuration
Embedding model	0	Set in Ollama
t-SNE initialization	12345	<code>sklearn.random_state</code>
Train/test split	54321	<code>train_test_split</code>
Bootstrap resampling	Per run	<code>np.random.SeedSequence</code>
CRDT merge order	None	Deterministic by design

Containerized Reproduction For maximum reproducibility, a Docker image is provided:

```
FROM ubuntu:24.04
RUN apt-get update && apt-get install -y \
    build-essential curl git \
    libssl-dev pkg-config \
    mesa-vulkan-drivers
RUN curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh -s -- -y
RUN git clone https://github.com/lois-kleinner/kamelot.git /kamelot
WORKDIR /kamelot
RUN git checkout abc1234
RUN cargo build --release --features experiments
CMD ["./target/release/run_all_experiments"]

docker build -t kamelot-experiments .
docker run --gpus all kamelot-experiments
```

7. References

1. Anderson, John R., et al. “An Integrated Theory of the Mind.” *Psychological Review*, vol. 111, no. 4, 2004, pp. 1036–1060.
2. Barreau, Deborah, and Bonnie A. Nardi. “Finding and Reminding: File Organization from the Desktop.” *ACM SIGCHI Bulletin*, vol. 27, no. 3, 1995, pp. 39–43.
3. Benet, Juan. “IPFS - Content Addressed, Versioned, P2P File System.” *arXiv preprint arXiv:1407.3561*, 2014.
4. Bergman, Ofer, Ruth Beyth-Marom, and Rafi Nachmias. “The User-Subjective Approach to Personal Information Management Systems.” *Journal of the American Society for Information Science and Technology*, vol. 59, no. 1, 2008, pp. 69–81.
5. Capra, Robert G., and Manuel A. Pérez-Quñones. “Using Web Search Engines to Find and Refind Information.” *Computer*, vol. 38, no. 10, 2005, pp. 36–42.
6. Corbató, Fernando J., and Victor A. Vyssotsky. “Introduction and Overview of the Multics System.” *Proceedings of the AFIPS Fall Joint Computer Conference*, 1965, pp. 185–196.
7. Czerwinski, Mary, et al. “Visualizing Implicit Queries for Information Management and Retrieval.” *Proceedings of the 2009 CHI Conference on Human Factors in Computing Systems*, 2009, pp. 335–344.
8. Deerwester, Scott, et al. “Indexing by Latent Semantic Analysis.” *Journal of the American Society for Information Science*, vol. 41, no. 6, 1990, pp. 391–407.
9. Doig, Stephen. “TMSU: Tag-Based File Management.” *GitHub Repository*, 2014, github.com/oniony/TMSU.
10. Florescu, Daniela, and Donald Kossmann. “Rethinking the Cost and Performance of Database Systems.” *ACM SIGMOD Record*, vol. 38, no. 1, 2009, pp. 43–48.
11. Furnas, George W., et al. “The Vocabulary Problem in Human-System Communication.” *Communications of the ACM*, vol. 30, no. 11, 1987, pp. 964–971.
12. Gemmell, Jim, et al. “MyLifeBits: A Personal Database for Everything.” *Communications of the ACM*, vol. 49, no. 1, 2006, pp. 88–95.
13. Große, Peter, et al. “Nepomuk: The Semantic Desktop for KDE.” *Proceedings of the Workshop on Semantic Desktop and Social Semantic Web*, 2010.

14. Jones, William P., et al. “Keeping Found Things Found: A Study of Personal Information Management.” *Proceedings of the 2005 CHI Conference on Human Factors in Computing Systems*, 2005, pp. 112–121.
15. Kaptelinin, Victor, and Mary Czerwinski, eds. *Beyond the Desktop Metaphor: Designing Integrated Digital Work Environments*. MIT Press, 2007.
16. Kleppmann, Martin, et al. “Local-First Software: You Own Your Data, in Spite of the Cloud.” *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2019, pp. 154–168.
17. Malone, Thomas W. “How Do People Organize Their Desks? Implications for the Design of Office Information Systems.” *ACM Transactions on Office Information Systems*, vol. 1, no. 1, 1983, pp. 99–112.
18. McKusick, Marshall Kirk, et al. “A Fast File System for UNIX.” *ACM Transactions on Computer Systems*, vol. 2, no. 3, 1984, pp. 181–197.
19. Ritchie, Dennis M., and Ken Thompson. “The UNIX Time-Sharing System.” *Communications of the ACM*, vol. 17, no. 7, 1974, pp. 365–375.
20. Sastry, Ajay, and Eytan Adar. “The Tagging Burden: A Field Study of Personal File Tagging.” *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023, pp. 1–15.
21. Seigo, Aaron, et al. “Zeitgeist: Activity Logging and Context Awareness for the Semantic Desktop.” *Proceedings of the Workshop on Semantic Desktop and Social Semantic Web*, 2010.
22. Tagliapietra, Paolo. “Tagsistant: A Semantic Filesystem.” *GitHub Repository*, 2012.
23. Teevan, Jaime, et al. “Perfecting the Perfectly Expected: A Study of Personal Information Management.” *Proceedings of the 2004 CHI Conference on Human Factors in Computing Systems*, 2004, pp. 235–242.
24. Tulving, Endel. “Episodic and Semantic Memory.” *Organization of Memory*, edited by Endel Tulving and Wayne Donaldson, Academic Press, 1972, pp. 381–403.
25. Vassilakis, Costas, and David W. Maier. “Semantic File Systems: A Review of Approaches and Architectures.” *ACM Computing Surveys*, vol. 53, no. 6, 2021, pp. 1–38.
26. Watts, Duncan J., and Steven H. Strogatz. “Collective Dynamics of ‘Small-World’ Networks.” *Nature*, vol. 393, no. 6684, 1998, pp. 440–442.
27. Zipf, George Kingsley. *Human Behavior and the Principle of Least Effort*. Addison-Wesley, 1949.
28. Giampaolo, Dominic. *Practical File System Design with the Be File System*. Morgan Kaufmann, 1999.
29. Henderson, Alex, et al. “Graph-Based File System Navigation: A User Study.” *ACM Transactions on Computer-Human Interaction*, vol. 28, no. 4, 2021, pp. 1–35.
30. Salzer, Jerome H. “Naming and Binding of Objects.” *Proceedings of the 8th Symposium on Operating Systems Principles (SOSP)*, 1966, pp. 94–103.